Bubble: Towards Scalable Evolving Graph Processing via Mini-Batch Sorting

Long Deng
University of Science and Technology of China
Hefei, China
ldeng@mail.ustc.edu.cn

Yongkun Li*
University of Science and Technology of China
Hefei, China
Anhui Provincial Key Laboratory of High Performance Computing
Hefei, China
ykli@ustc.edu.cn

Zaigui Zhang
Jinan Inspur Data Technology Co., Ltd.
Jinan, China
zhangzg@inspur.com

Yinlong Xu
University of Science and Technology of China
Hefei, China
ylxu@ustc.edu.cn

John C. S. Lui
The Chinese University of Hong Kong
Hong Kong, Hong Kong
cslui@cse.cuhk.edu.hk

Abstract

Evolving graph processing has become a critical component in various applications and is gaining increasing attention. However, existing evolving graph systems suffer from cache contention and workload imbalance between threads, which leads to poor scalability and performance degradation on modern multi-core computers.

In this paper, we introduce Bubble, a high-performance evolving graph processing engine designed with high scalability. By employing a novel graph format based on mini-batch sorting, Bubble utilizes the private caches of modern processor cores, achieving near-linear scalability in graph ingestion, while maintaining high performance for graph analytics. Compared with state-of-the-art systems, including LSGraph, GraphOne, and XPGraph, Bubble achieves 2.46×-8.86× higher throughput in ingestion and 0.77×-3.29× the performance when running common graph algorithms.

CCS Concepts

• **Information systems** → **Data structures**; **Data management systems**.

Keywords

Evolving graph, In-memory graph processing, Scalability

ACM Reference Format:

Long Deng, Yongkun Li, Zaigui Zhang, Yinlong Xu, and John C. S. Lui. 2025. Bubble: Towards Scalable Evolving Graph Processing via Mini-Batch Sorting. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3712285.3759897

*Corresponding author.

SC '25, St Louis, MO, USA

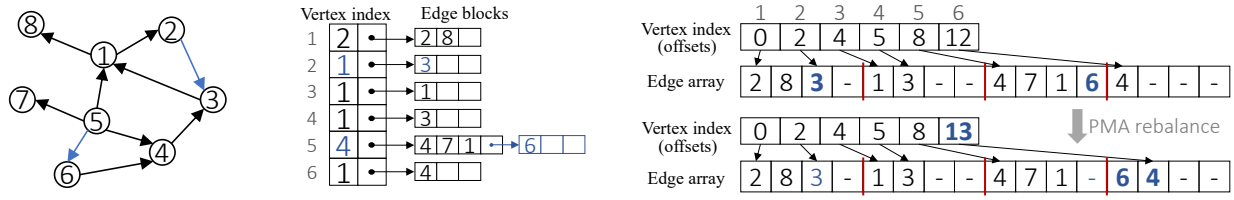
ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759897

1 Introduction

Graph processing is receiving increasing attention because of its broad usage in social networks [55], recommendation systems [71], and artificial intelligence [50]. Real-world graphs are often evolving graphs [6, 11, 22], which are changing rapidly, requiring graph systems to handle both high-throughput ingestion (i.e., graph updates) and efficient analytics. Current in-memory evolving graph systems refine existing graph formats such as adjacency lists [29, 31, 39, 72] and compressed sparse row (CSR) [23, 30, 51] or introduce new ones such as learned indexes [61] and packed memory array (PMA) [23, 34, 58] to balance ingestion and analytics performance.

As processor cores continue to increase, a single processor can feature dozens or even hundreds of cores [1, 3], requiring graph processing systems to have parallel capabilities to take advantage of powerful computing resources. While parallel graph analytics has achieved significant progress [48, 56, 66, 73, 78], ingestion in state-of-the-art evolving graph systems still uses a simple *vertex-centric partitioned parallel strategy*: the graph is divided into non-overlapping subgraphs (or partitions) by vertex ranges, and the graph updates are processed in batches, new edges in each batch are dispatched to the corresponding partitions, and finally, partitions are updated in parallel, with each partition being processed by a single thread [39, 58, 61, 72].

This strategy works well under low parallelism, as it prevents data conflicts caused by multiple threads updating the same partition, thus eliminating synchronization overhead. However, our experiments show that when these systems scale to dozens of threads, their ingestion throughput grows slowly or even decreases. For instance, we measured the ingestion throughput of three state-of-the-art systems (GraphOne [39, 40], XPGraph [72], and LSGraph [61]) on the Twitter dataset. When the thread count doubles from 40 to 80, the ingestion throughput of LSGraph increases by only 32.1%, and GraphOne and XPGraph drop by 5.1% and 15.2%, respectively.



(a) An example graph. (b) Block-based adjacency list. (c) CSR, with a PMA as the neighbor array.

Figure 1: Common Dynamic Graph Formats. Blue parts indicate newly added edges and modified data.

Moreover, LSGraph also stops scaling when the thread count exceeds 64. As hardware continues to scale, these issues can hamper the adoption of evolving graph processing on future processors.

To understand why performance degrades with more cores, we analyze the components of existing systems and identify three key scalability bottlenecks: (1) Cache contention. When multiple partitions access the last-level cache (LLC) simultaneously, they frequently evict the data of other partitions, which increases memory latency by up to 3.2× in our evaluations; (2) Load imbalance. Due to the skewness of graph data, some threads finish their tasks early and must be idle to wait for others, causing CPU utilization to drop below 50%; (3) Dispatching overhead. Excessive partitioning leads to high dispatching overhead, which reduces dispatching throughput at 64 threads to half of that at 8 threads, becoming a new bottleneck for graph ingestion.

In this work, we explore the design of highly scalable evolving graph systems and try to address the aforementioned three issues. To this end, we exploit the potential of existing architectures, leveraging private caches (L1 & L2 cache) to reduce cache contention. We propose a new graph format called MuSEA, which converts the graph into many locally sorted segments (i.e., *mini-batch sorting*) so as to perform larger merges, achieving higher scalability. We also propose a new parallel strategy with a novel dispatcher and a work-stealing-based load balancing mechanism to mitigate load imbalance and prevent dispatching from becoming a bottleneck.

In summary, our main innovations include:

- We propose MuSEA, a storage format based on *mini-batch sorting*, which mitigates cache contention among multiple cores by leveraging private caches (L1 & L2 cache), allowing many partitions to be updated in parallel without sacrificing per-partition throughput.
- We design a new parallel ingestion policy that divides threads into two categories: dispatchers and workers. A lock-free dispatcher structure enables multiple dispatchers to distribute updates simultaneously, preventing dispatch overhead from being a bottleneck. Worker threads, bound to each partition, process local updates and can help others via a work-stealing policy to improve overall performance.
- We implement multiple graph query optimizations in the multi-segment ordered structure of MuSEA. These optimizations allow access to edges stored in MuSEA with customizable patterns to adapt to the memory access requirements of different graph algorithms, thereby improving memory access efficiency and reducing the running time of graph analytics.

Combining these designs, we implement Bubble, a scalable evolving graph system for modern multicore processors. Bubble provides general APIs for graph updates and queries, supporting diverse dynamic graph workloads with minimal code modifications. Our experiments on multiple datasets demonstrate that Bubble outperforms state-of-the-art evolving graph systems by 2.46×-8.86× in ingestion throughput and achieves 0.77×-3.29× the performance when running common graph algorithms.

The rest of this paper is organized as follows. In §2, we introduce background knowledge for evolving graph systems, including common storage formats and parallel ingestion strategies, then analyze their limitations. In §3, we present design details of Bubble, covering the MuSEA format, ingestion strategies, and analytics optimizations. §4 evaluates performance of Bubble on diverse datasets and workloads and compares it to state-of-the-art systems. §5 discusses related works, and §6 concludes this paper.

2 Background and Motivation

2.1 Dynamic Graph Formats

An evolving graph system (also called a dynamic graph system) needs to support both high-throughput graph ingestion (i.e., graph updates) and efficient graph queries. Most of the existing evolving graph systems are based on the two common graph storage formats, adjacency list and compressed sparse row (CSR). Figure 1 presents these two graph formats, and their common optimizations for dynamic graph workloads.

Adjacency list. The adjacency list is the most widely adopted format for evolving graph workloads [20, 31, 69, 76]. It uses a vertex index array containing pointers to each vertex's neighbor list. To reduce the frequently memory allocation and pointers chasing of basic linked list implementations, modern systems employ block-based structures where neighbors are stored in memory blocks which has better memory locality and can reserve space for incoming updates [16, 27, 39, 40, 47]. Alternatively, some systems implement dynamic arrays that resize by doubling capacity and relocating neighbors when needed, balancing insertion efficiency with memory usage [21, 29, 72].

Compressed sparse row. A compressed sparse row, or CSR, stores all neighbors sequentially in a contiguous array while maintaining a vertex index array that points to the start of each vertex's neighbors. This design offers excellent cache locality for static graph analytics but struggles with dynamic updates since inserting edges requires shifting all subsequent elements. Modern evolving

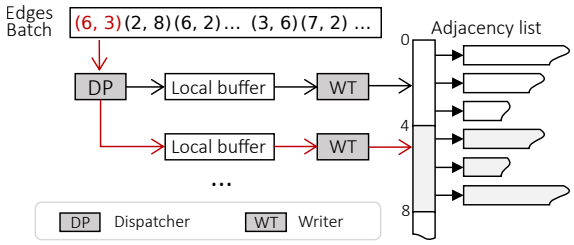


Figure 2: Parallel Ingestion by Vertex-Centric Partitioning.

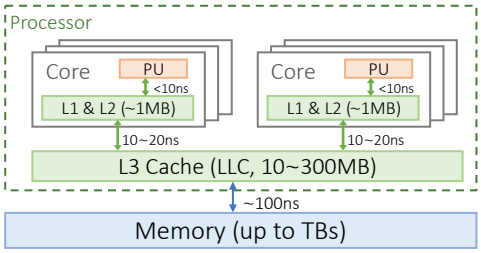
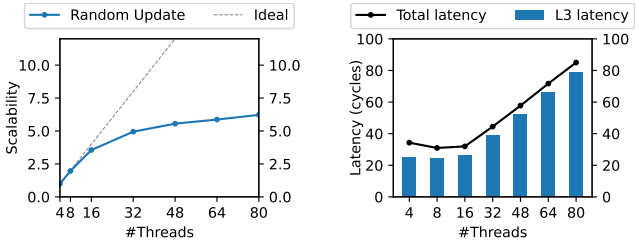


Figure 3: Cache Hierarchy of a Typical Multi-Core Processor.



(a) Random update throughput with different numbers of threads.

(b) Average update latency and L3 stall time (in cycles).

Figure 4: Scalability and Latency of Vertex Index Insertions.

graph systems commonly replace the neighbors array in CSR with a Packed Memory Array (PMA) [5, 10]. PMA maintains sorted elements with intentional gaps between them, allowing new edges to be inserted into these pre-allocated spaces. When gaps in a memory region become insufficient, PMA moves elements to keep the density of regions balanced, and if the overall density exceeds a threshold, it reallocates a larger array. This approach preserves CSR's sequential access benefits while supporting efficient updates [23, 30, 34, 51, 53, 65].

Other graph formats. Some systems employ tree structures or hybrid data structures to enhance functionality or performance [25, 35, 58, 61]. For instance, Aspen [35] utilizes an Adaptive Radix Tree (ART) [44] to support multiple historical time window queries, while LSGraph [61] combines PMA and tree structures to optimize performance for vertices of varying degrees. Despite their complexity, these formats follow a similar design principle to adjacency lists and CSR, where graph data is managed as two parts: an edges (or neighbors) storage and a vertex index pointing to the storage. By searching and updating the vertex index, new edges can be easily inserted into the correct position of the edges storage, and neighbors of each vertex can also be fetched efficiently.

2.2 Parallel Ingestion of Evolving Graphs

Today's processors feature dozens or even hundreds of cores, making it necessary for systems to use multi-threading to effectively use all available computing power. However, in evolving graph processing, edges often arrive randomly, causing parallel processing by multiple threads to create conflicts when writing to the vertex indexes and the edges storage. While out-of-core systems typically use per-vertex locks to maintain data consistency [23, 31, 51, 65, 81], this lock-based approach severely restricts ingestion throughput for in-memory systems [27, 47].

For higher ingestion speeds, recent in-memory systems batch updates and use a *vertex-centric partitioning strategy* to ingest edges of a batch in parallel [25, 39, 47, 58, 61, 72]. Figure 2 illustrates a typical vertex-centric partitioning strategy, where the graph is divided into partitions, each containing a specific range of vertices. To process a batch of edges, a dispatcher sends edges based on their source vertices to the corresponding partition buffers. Since the edges in each buffer are related to a distinct range of vertices, this design allows each partition to be updated independently by a writer thread, eliminating the need for locks or atomic operations.

Despite similar design principles, a key difference among these systems is the design of the dispatcher. For example, GraphOne [39] and XPGraph [72] deploy a single-thread dispatcher, which scans the whole batch and distributes edges to corresponding buffers, while LSGraph [61] and Terrace [58] parallelize sorting the edges in the batch and then split the batch into ranges by edges' source vertex. We will revisit different dispatcher designs and their impact on parallel loading performance in §2.3.3.

2.3 Limitations of Existing Works

Although the aforementioned solution performs well with a small number of threads, we observed that last-level cache contention, load imbalance, and the bottleneck of the dispatcher prevent the system from scaling to more cores. In this section, we will analyze these limitations in detail.

2.3.1 LLC contention. Figure 3 shows the memory hierarchy of a typical enterprise processor. Cache levels closest to the processing units, such as L1 and L2 caches, are core-private, allowing each core to access its own cache without contention. In contrast, the last-level cache (LLC, or L3 cache in common architectures) and main memory are shared among cores. When cache capacity is insufficient, data accesses from one core may evict other cores' data from the LLC, causing cache contention even without direct data conflicts.

In vertex-centric partitioning, each graph update modifies the vertex index and edge storage. For large graph workloads, the sizes of vertex sets and edge sets often exceed typical LLC capacities, leading to frequent L3 cache misses and random DRAM accesses. Existing systems batch updates to reduce the impact of edge writing, but they often ignore the cache contention caused by vertex index updates. We observe that random updates of vertex indexes suffer serious performance issues when scaling to tens of threads. Figure 4a shows the throughput of random vertex index updates with

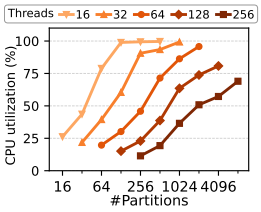


Figure 5: CPU Utilization with Different Writers Count.

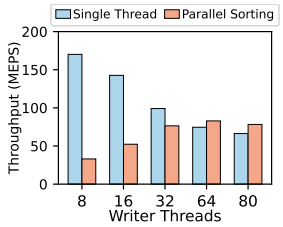


Figure 6: Throughput of Two Types of Dispatchers.

different thread counts. Although the throughput grows linearly when the number of threads is small, it reaches a bottleneck after 32 threads. We further analyze the latency and L3 cache stalls of vertex index updates. As shown in Figure 4b, most of the insertion latency is spent on L3 stalls, and these stalls increase by 3.2× when the number of threads increases from 16 to 80, which indicates that the contention of L3 cache is the main reason for poor scalability.

2.3.2 Workload Imbalance. Real-world graphs are often skewed, causing incoming edges in evolving datasets to be unevenly distributed. During batch ingestion, skewness leads to imbalanced workloads among writer threads: threads that finish early remain idle while waiting for heavily loaded ones, resulting in inefficient CPU utilization. Existing systems address this by creating more partitions than threads and assigning leftover partitions to those that finish early [39]. However, as the number of threads increases, the skewness intensifies, and simply increasing the number of partitions becomes insufficient for effective workload balancing.

To illustrate this, we ran a simulation experiment on LiveJournal to examine how the number of threads and partitions affects CPU utilization. As shown in Figure 5, with fewer threads, increasing the number of partitions can achieve nearly 100% CPU utilization. However, as the number of threads increases, the CPU utilization drops significantly. GraphOne and XPGraph both use a maximum of 256 partitions, leading to less than 50% CPU utilization at 64 threads. Splitting the graph into more partitions is impractical (e.g., 2048 partitions with 128 threads) because excessive partitioning reduces the average edges per partition, increases scheduling overhead, and makes the ingestion performance even worse. In conclusion, load imbalance still restricts system performance under high thread count.

2.3.3 Dispatching Overhead. As mentioned in §2.2, two types of dispatchers are commonly used in existing systems. The single-thread dispatcher (ST) uses an extra dispatcher thread to copy each edge to the corresponding partition buffer. In contrast, the parallel-sorting dispatcher (PS) employs all threads to first parallelly sort all the edges in a batch, and then splits the sorted edges into ranges for each writer.

We compare the performance of these two dispatcher types with different numbers of writers (with a fixed partition count of 4× the number of writers). As shown in Figure 6, when the number of writers is small, the ST dispatcher writes to fewer partitions, resulting in better cache affinity and throughput exceeding 150 MEPS. However, as the number of writers increases, the ST dispatcher needs to write to more partitions, leading to reduced cache and prefetching efficiency, and consequently lower throughput. In contrast, the PS dispatcher utilizes all threads, so its performance improves with higher thread counts. Yet, due to contention and scheduling overhead during parallel sorting, the PS dispatcher's performance growth slows after 32 threads and even starts to decline. At 80 threads, both dispatchers' throughputs fall below 80 MEPS, indicating that dispatching overhead becomes a potential bottleneck for the evolving graph system.

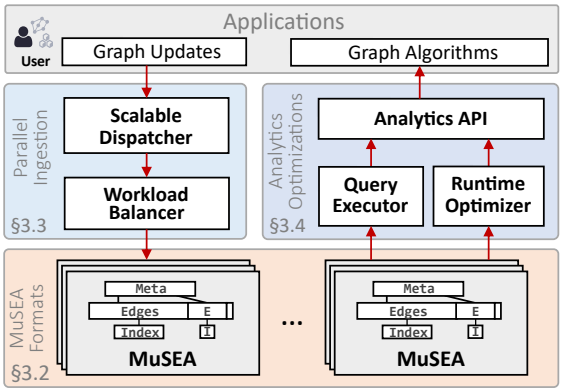


Figure 7: Overview of Bubble.

3 Bubble Design

3.1 Main Idea and Challenges

Main Idea. The most critical factor that limits the scalability of graph systems is LLC contention (see §2.3.1). However, due to the huge size of graph data, it is not practical to completely avoid LLC contention, instead, it is more important to mitigate the impact of LLC contention. We observe that the private cache (i.e., L1 and L2 cache) presents an opportunity for this. As shown in Figure 3, most modern processors have a large amount of private cache. Owing to the non-inclusive cache hierarchy of modern processors [67], even if a piece of data is evicted from the LLC, the same copy in the private cache remains preserved. Therefore, we refer to a routine that only accesses data in private cache as *contention-safe*, meaning that it can run parallelly without being affected by other threads using LLC (and without taking up the LLC and affecting other threads).

Therefore, to improve the scalability of graph ingestion and allow a large number of partitions to update in parallel while minimizing cross-partition interference, it is essential to ensure that most of the graph update process is contention-safe. To achieve this, our main idea is to convert graph updates into multiple local sorts and a small number of global merges. Specifically, we divide newly arrived edges in a partition into mini-batches that fit within L2 cache, sort each mini-batch to create sorted segments (mini-batch sorting), and periodically merge multiple sorted segments into a larger one to maintain global ordering. Because each mini-batch sort is contention-safe, the partition being sorted is unaffected by other partitions (even if they are merging), preserving high ingestion throughput as more partitions are used.

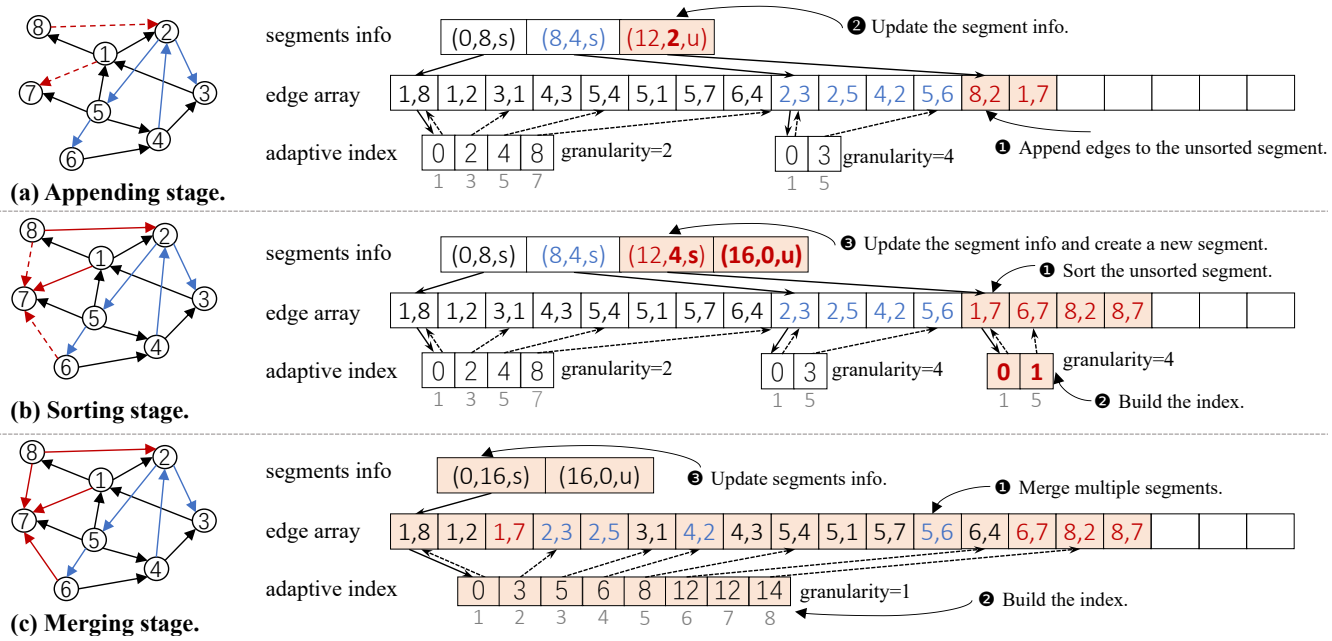


Figure 8: MuSEA (Multi-Segments Edge Array) Formats and Operations. Segments info represented as (offset, length, state), e.g., (8,4,s) indicates a segment from position 8 in the edge array, with 4 sorted edges in it (s: sorted, u: unsorted). Granularity represents the interval of vertices with records in the index.

Challenges. To build a complete system based on the above idea, we mainly face three challenges. First, merging multiple sorted segments is heavy and may become the new bottleneck of the system. Second, before batches are merged, edges of the same vertex are scattered across different mini-batches, making it difficult to find the neighbors of a vertex efficiently, which may affect the analytics performance. Third, the workload imbalance and dispatcher bottleneck described in §2.3.2 and §2.3.3 should be addressed, to ensure all partitions have enough updates to process.

3.2 Overview

To overcome the aforementioned challenges and achieve a highly scalable evolving graph system, we propose Bubble. As Figure 7 illustrates, to efficiently process both graph updates and graph algorithms, Bubble consists of three key components: the MuSEA graph format, a parallel ingestion policy, and optimizations for graph analytics. Similar to existing works, graphs are partitioned into subgraphs by vertex ranges. In Bubble, each subgraph is stored as a MuSEA, a novel format that supports efficient parallel updates. All new edges are dispatched to the corresponding MuSEA to keep the workload balanced under the ingestion policy. Graph algorithms are implemented in Bubble 's vertex-centric API, with additional optimizations to enhance their performance. To better understand the design of Bubble, we first briefly introduce these three components, and then provide detailed explanations in the following subsections.

MuSEA Format. The core of Bubble is MuSEA (§3.3), a novel graph format, which adopts mini-batch sorting and periodic merging to utilize the private cache of CPU cores, allowing the system to update dozens of partitions in parallel without interference. Moreover, though the neighbors of a vertex are stored discontinuously, MuSEA supports efficient vertex-centric queries by maintaining multiple adaptive indexes that can quickly locate a vertex's edges.

High-parallel Ingestion Policy. To reduce workload imbalance and dispatching overhead, Bubble employs a new parallel ingestion policy. This policy includes a scalable multi-core dispatcher and a work-stealing-based load-balancing mechanism. The carefully designed lock-free structures in the multi-core dispatcher allow multiple threads to append updates to the same partition simultaneously. The work-stealing mechanism enables idle writer threads to assist with updates in busy partitions, addressing workload imbalance due to skewed graphs. We detail these designs in §3.4.

Analytics Optimizations. Finally, Bubble supports efficient graph analytics through both a general vertex-centric API and query-optimized APIs designed for common graph access patterns. Bubble converts query requests into sequential scans of MuSEA when possible and caches some query results within an algorithm execution to mitigate the overhead caused by discontinuous neighbor storage. We will discuss these optimizations in §3.5.

3.3 MuSEA Format

Bubble stores each subgraph as a *multi-segment edge array*, or MuSEA. To address LLC contention, MuSEA uses mini-batch sorting to ingest updates. In this section, we first introduce basic structure of MuSEA (§3.3.1), followed by its operations, including updates (§3.3.2 and §3.3.3) and queries (§3.3.4).

3.3.1 Data structure. Figure 8(a) illustrates the structure of MuSEA. The core of MuSEA is an edge array containing all edges in the subgraph. The array is divided into multiple segments (shown in different colors), with newer segments holding recently inserted edges. Except for the last segment, edges in each segment are sorted by source vertex ID. However, edges across segments are unordered, so different segments can store edges for the same vertex. A metadata array records each segment's start position, length, and sorted status. Each sorted segment is associated with an index array storing offsets that point to the first edge of some vertices in the segment.

To reduce memory overhead, an index does not include offsets for all vertices. Instead, the index is adaptive: it first sets the index size based on the segment's edge count, then determines the index granularity from both this size and the total vertex count in the partition, storing only one vertex offset per granularity. For example, in Figure 8(a), the first segment has an index size of 4 (half the segment length). Because there are 8 vertices in the partition, the index granularity is $8/4 = 2$, so the index stores offsets for every two vertices (i.e., for vertices 1, 3, 5, and 7).

3.3.2 Sorting-based Updates. As shown in Figure 8, the update process of MuSEA consists of three stages: appending, sorting, and merging.

Append. Figure 8(a) shows the process of the appending stage. In this stage, edges are directly appended to the end of the edge array, specifically in the last unsorted segment, and the length in the segment info is updated accordingly. The append operation is a sequential write to memory, which is contention-safe and can fully utilize memory bandwidth.

Sorting. Simply appending edges to the end of the array provides high update throughput, but the unordered edge data limits graph analysis performance. Therefore, when the number of edges in the unsorted segment exceeds a set threshold, the sorting stage creates a new sorted segment. As shown in Figure 8(b), edges in the segment are first sorted, then an adaptive index is created and linked to the new sorted segment. (see §3.3.1 for details of index, an index can be constructed by simply scanning the segment once.) Finally, the metadata record of the segment is updated, and an empty unsorted segment is set at the end of the array for subsequent new edge writes. By setting an appropriate threshold, each sorting segment can fit within L2 cache, allowing multiple MuSEA structures to sort simultaneously without causing LLC contention.

Merging. The sorting stage generates numerous segments, which can degrade performance during graph queries. To reduce the number of segments, it is necessary to periodically merge them. Figure 8(c) illustrates the process of merging three segments from graph (b). The merging process is similar to the sorting stage: first, edges in all merging segments are reordered to form a new sorted segment, then the original indexes of the segments are released, and a new index is created. Finally, the metadata records of the merged segments are combined into a single record. The merging stage involves extensive edge rewriting, and frequent merges can lead to significant performance degradation. In the next section, we will further analyze the overhead of the merging stage and introduce how to mitigate its impact on system performance through a proper merging policy.

3.3.3 Merging Policy. The merging stage aims to minimize the number of segments while controlling the total amount of data rewriting (i.e., write amplification). Consider a merge operation that merges m segments with lengths $L_1, L_2, \ldots, L_m$, we define its *merging efficiency* as:

$$E_{\text{merge}} = \frac{\sum_{i=1}^{m} L_i}{\max(L_1, L_2, \ldots, L_m)} \tag{1}$$

In other words, merging efficiency is the total length of all segments divided by the length of the longest segment. To illustrate this intuitively, suppose a longer segment is merged sequentially with multiple shorter segments. Each time a shorter segment is merged, all data in the longer segment must be rewritten, yet the length of the merged segment does not grow much. Compared to first merging all the shorter segments into one larger segment, and then merging the new segment with the longer segment, the sequential approach introduces more write amplification and is therefore less efficient.

Next, to show how controlling the efficiency can bound overall write amplification, assume every merge operation satisfies $E_{\text{merge}} \geq \alpha$. For any edge in the graph, consider the length of the segment it belongs to. By definition, after each merge, the segment containing that edge grows by at least a factor of α. Thus, after K merges, its length is at least $\alpha^K L$, where L is the initial length of the segment. Because each merge rewrites data in the merged segments exactly once, any edge in a MuSEA with N edges is rewritten at most $\log_\alpha(N/L)$ times, so the upper bound of total write amplification is $\log_\alpha(N/L)$.

Segments Selection for Merging. Based on the above analysis, we propose a merging strategy guided by merging efficiency. Its principles are: (1) ensuring the efficiency of each merge operation is above a preset threshold α, and (2) merging as many segments as possible. After each sorting stage, the system calculates the merging efficiency of the last K segments in sequence (K from 1 to the count of segments in the partitions) and selects the largest K that satisfies the condition. If no K meets this requirement, no merging is performed after this sorting stage. Considering the total length of all segments in MuSEA, we can prove that as the number of segments grows, there always exists a K that meets the condition. Thus, the number of segments in MuSEA does not exceed $(\alpha - 1)\log_\alpha(N/L)$. In Bubble, we set $\alpha = 2$ as the default threshold to balance read and write efficiency. This setting yields an upper bound on both the number of segments and the write amplification of $\log_2(N/L)$. We further discuss system performance under different α values in the evaluation (§4.6).

3.3.4 Basic Graph Query. Though the edges of a vertex may be scattered across multiple segments, MuSEA can still provide efficient vertex-centric query performance because edges within each segment are sorted and indexed. To query the neighbors of a vertex (or the edges from this vertex), one must sequentially check each segment. For ordered segments, the system first uses the adaptive index to locate the general range of edges that likely contain those relevant to the vertex, and then applies a binary search within that range. For unordered segments, because the required edges may appear anywhere, the entire segment must be scanned.

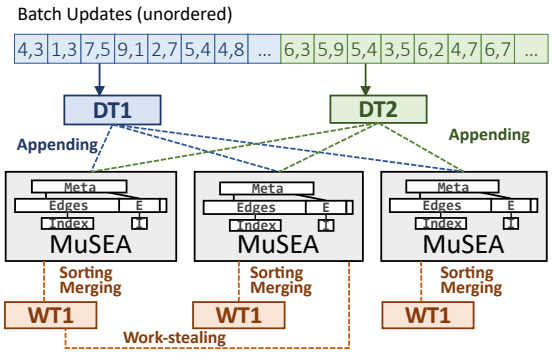


Figure 9: Parallel Ingestion Policy and Threads Management.

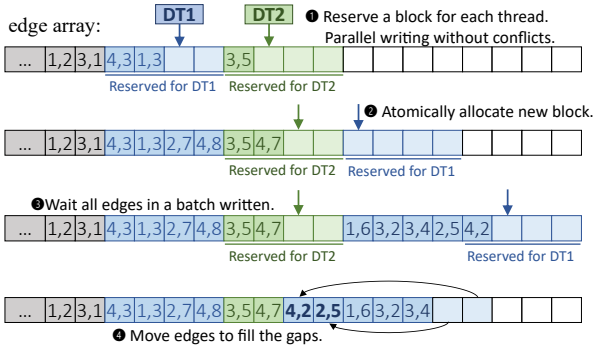


Figure 10: Parallel Appending Edges into Single Edge Array.

Although this lookup process is more involved than in common graph formats like CSR or adjacency lists, it does not severely degrade query performance. First, because the adaptive index keeps the general range small (typically within a cache line), the binary searching in the range incurs minimal overhead. Second, as with adjacency lists, edges of the same vertex are not strictly contiguous, but MuSEA's merging strategy limits the total number of segments, often yielding better performance than adjacency lists when querying high-degree vertices. Furthermore, while querying a vertex may involve scanning some unordered segments, our experiments show that by selecting an appropriate sorting threshold, scanning unordered edges accounts for less than 5% of the total query time. Finally, for certain common graph access patterns, Bubble provides some optimizations, detailed in §3.5.

3.3.5 Algorithmic Details. MuSEA's update process requires significant sorting and merging operations. Therefore, selecting suitable algorithms for these tasks is important for overall system performance. In this work, we describe the algorithms we use and explain our choices.

A proper sorting algorithm for MuSEA should (1) be in-place to avoid extra LLC accesses, (2) have good locality to make use of the L1 cache, and (3) perform well on short random sequences. We choose PDQSort (Pattern-defeating Quicksort) [59] because it is fast on short random sequences. Replacing it with another in-place sorting algorithm does not affect scalability of Bubble, but a slower algorithm may reduce update throughput.

For merging, traditional merge algorithms require extra memory to temporarily store all merged edges and then move them back, which causes additional LLC accesses. To avoid this, TimMerge, the merge routine from TimSort [60], is employed. TimMerge buffers only a small run and performs multi-segment merging with O(n) additional writes. Applied to MuSEA, this approach is both faster and more memory-efficient than straightforward merge algorithms.

3.4 Highly Parallel Batch Ingestion

By following the principle of mini-batch sorting, the MuSEA format allows the system to maintain tens or hundreds of subgraphs and perform parallel updates with high throughput. However, to achieve highly scalable ingestion, it still needs to address the other two issues mentioned above: the workload imbalance caused by skewed graphs and the overhead of dispatching. In this section, we will introduce the design of the parallel ingestion policy in Bubble, which includes a load-balancing mechanism based on work-stealing and a dispatcher that supports parallel dispatching. Figure 9 shows the parallel ingestion strategy and threads management in Bubble. Two types of threads are used in ingestion: dispatcher threads (DT) read the updates and append edges to the corresponding MuSEA, while each MuSEA is bound to a worker thread (WT), which is responsible for sorting and merging in the MuSEA.

3.4.1 Scalable Dispatcher. To avoid potential performance bottlenecks in dispatching, allowing multiple threads to dispatch updates in parallel is crucial. However, as mentioned in §2.3.3, multiple dispatcher threads may need to update the same partition simultaneously, requiring additional synchronization that could lead to further performance degradation. To address this, Bubble introduces a lock-free, multi-threaded dispatcher. Figure 10 illustrates how multiple threads append edges to the same edge array in parallel. First, at the beginning of batch ingestion, each thread atomically reserves a block with several empty edge slots at the end of the edge array, allowing the threads to safely append edges into the block without synchronization. If any thread's block becomes full, another block is allocated for it. After all threads finish dispatching, there may be some empty slots in the edge array. A single thread then reverse-scans the edge array and moves edges from the end to fill these empty slots, ensuring that all edges are contiguous. Meanwhile, by reading each thread's write position, worker threads can identify which part of the edge array is filled and correctly perform sorting and merging without waiting for all edges in the batch to be dispatched.

3.4.2 Load Balancing by Work-stealing. Another issue affecting ingestion scalability is load imbalance. A worker thread managing a hot partition may be blocked by the sorting and merging, while other worker threads are idle. In existing evolving graph systems, it is difficult for idle threads to help busy ones because concurrent insertions to the same partition introduce data conflicts, resulting in additional synchronization overhead. In contrast, although the merging stage in MuSEA modifies extensive metadata and is also difficult to parallelize, the sorting step exhibits good locality. Thus, when one worker thread is busy merging, idle workers can help by sorting newly appended edges. In our design, if a worker thread

remains idle for a while, it checks the status of other partitions and assists the one with the most unsorted edges to create a sorted segment.

3.5 Analytics Optimizations

While MuSEA can provide efficient vertex-centric queries (§3.3.4), the process remains more complex than in common graph formats such as CSR or adjacency lists. To further improve graph analytics performance, Bubble implements several optimizations for common graph access patterns. All of these optimizations are implemented as high-level, vertex-centric APIs in Bubble, enabling users to apply them easily in their algorithms.

Horizontal scanning. Many graph algorithms, such as PageRank, SSSP, and CC without shortcuts, require scanning all edges. For these algorithms, there is no need to query each vertex's neighbors one by one, which would otherwise require traversing all segments for every vertex. Instead, Bubble supports horizontal scanning, allowing users to read the segments in sequence and skip unnecessary binary searches. This optimization is also handy for range queries, where users specify a range of vertex IDs and want all neighbors in that range. In this case, Bubble accesses the index and performs a single binary search per segment, after which edges are read by sequentially scanning the segments.

Multi-head iterators. Another category of algorithms must retrieve neighbors in a particular order. For instance, triangle counting (TC) needs to scan neighbors by their target vertex ID. Although Bubble can sort edges in different ways (see Bubble-O in §4.1 for details), merging edges between segments still requires checking each segment. To accelerate this, Bubble provides multi-head iterators, which store a pointer to each segment and move these pointers under specific conditions. For example, to iterate edges by target vertex ID, the pointers are placed in a min-heap, and the algorithm always processes the pointer at the top of the heap. This optimization also benefits algorithms like CC with shortcuts, where only the first N neighbors are needed, and these pointers help determine whether enough neighbors have been found across all segments.

In-analytics cache. Certain queries that are straightforward in CSR or adjacency lists may incur higher overhead in Bubble. For example, querying the degree of a vertex in Bubble requires a binary search for each segment, which can be costly in algorithms that repeatedly access vertex degrees (e.g., PageRank). To address this, Bubble provides an in-analytics cache that allows users to store some query results within the algorithm, thus speeding up such accesses. Additionally, we implement a pre-calculation cache, which enables users to precompute vertex degrees (and similar information) in advance. Compared to calculating them during the algorithm, pre-calculation typically achieves higher speedups by leveraging the aforementioned optimizations and offers better locality.

4 Evaluation

4.1 Experimental Setup

Environment. All experiments are conducted on a dual-socket server with two Intel Xeon Gold 5218R CPUs at 2.10 GHz. Each processor has 20 physical cores and 40 logical cores with hyper-threading enabled. Each CPU has 28 MB of L3 cache. The L2 and L1 caches are 1 MB and 64 KB per core, respectively. The server is equipped with 220 GB of main memory and 1.5 TB of NVMe SSD storage. The operating system is Ubuntu 20.04.6 LTS with kernel version 5.15.0.

Table 1: Statistics of the Datasets.

Datasets	Domain	Vertices	Edges
LiveJournal (LJ)	social	4,847,571	68,993,773
Protein (PR1)	biology	8,745,542	1,058,120,062
Twitter (TW)	social	61,578,415	1,468,365,182
Friendster (FR)	social	68,349,466	2,586,147,869
UK-2007 (UK)	web	110,123,614	3,944,932,566
Protein2 (PR2)	biology	17,491,086	4,234,728,014

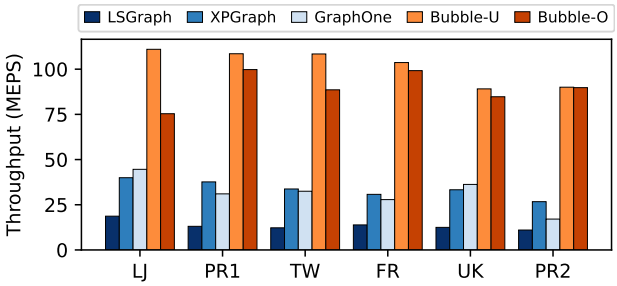


Figure 11: The Throughput of Graph Ingestion.

Comparison systems. We compare Bubble with three state-of-the-art evolving graph systems: LSGraph [61], XPGraph [72], and GraphOne [39, 40]. These systems use distinct data structures to balance updates and analytics performance: LSGraph employs tree-structured PMAs, XPGraph uses adjacency lists with dynamic arrays, and GraphOne uses block-based adjacency lists. GraphOne offers high performance for both ingestion and analytics, making it a strong baseline. Although XPGraph was originally designed for persistent memory, we use its DRAM-only version as a baseline due to its superior insertion throughput. LSGraph is a state-of-the-art system that guarantees ordered neighbors [61], which is important for workloads such as triangle counting or graph pattern mining [13]. Since the original LSGraph ingests edges only in a single direction and lacks support for querying in-neighbors, we implement a two-way version that maintains separate graphs for both directions, ensuring a fair comparison.

Two versions of Bubble are used in our evaluations: Bubble-U, which sorts edges by source vertex ID, and Bubble-O, which sorts edges by both source and target vertex IDs. As a result, Bubble-U is more efficient for ingestion, while Bubble-O supports queries requiring ordered neighbors.

Datasets. We evaluate Bubble on six real-world graphs from different domains [15, 41, 45]. Table 1 lists the key statistics of these datasets. These graphs are widely used for benchmarking graph systems. LiveJournal [8], Twitter [42], and Friendster [41] are online social networks that follow a power-law distribution. UK-2007 [14], being a web graph, exhibits a similar power-law degree distribution to Twitter, though it has fewer extremely high-degree vertices. Protein and Protein2 [7] are from biology research, featuring higher

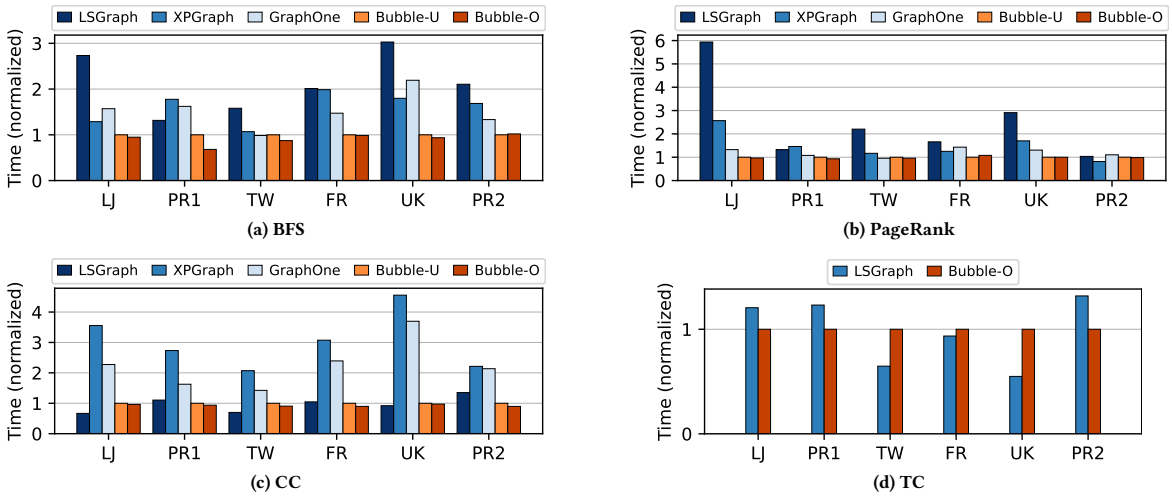


Figure 12: Time to Run Graph Algorithms, Normalized to Bubble-U.

density but less skewed distributions compared with social networks. All datasets are converted to binary edge lists with 4-byte vertex IDs and randomly shuffled before evaluation.

Graph Analytics Workloads. We adopt four common graph computing algorithms to evaluate graph analytics performance: breadth-first search (BFS), PageRank (PR), connected component (CC), and triangle counting (TC).

We implement each algorithm on all systems following the implementations GAP Benchmark Suite [9]. Every effort is made to ensure that the compared algorithms are consistent, and the results are verified for a fair comparison. LSGraph only provides a vertex-centric programming model based on `EdgeMap` and `VertexMap` operations [66], which differs from the shared memory models the other three systems provide. The algorithms are carefully translated into a version that can run on LSGraph, and for some algorithm operations that are difficult to express using `EdgeMap` and `VertexMap`, the underlying LSGraph interface is called to ensure that the algorithms remain completely consistent across all systems.

4.2 Graph Ingestion Performance

We first evaluate the graph ingestion performance of different systems in MEPS (Million Edges Per Second) on all the datasets listed in Table 1. To properly measure the ingestion throughput of systems with asynchronous updates (e.g., GraphOne), we record the time until the last edge is visible for analytics. This means all edges have been applied in GraphOne and XPGraph, and all sorting and merging operations are finished in Bubble. We use 80 threads for all tested systems to utilize all CPU cores on our machine. The ingestion batch size is set to 65536 (i.e., 2^{16}), following the configuration recommended by GraphOne and XPGraph. Datasets are loaded into memory before testing to avoid I/O overhead.

Figure 11 shows the ingestion throughput of each system. Bubble-U and Bubble-O outperform the other systems on all datasets. In particular, the throughput of Bubble-U reaches 90.03 MEPS on the largest dataset (PR2), achieving speedups of 5.94×–8.86×, 2.68×–3.37×, and 2.46×–5.27× over LSGraph, XPGraph, and GraphOne, respectively. The ordered-neighbors version (Bubble-O) is slightly slower than Bubble-U, at 67.9%–99.7% of Bubble-U's throughput, but still achieves 2.54×–8.15× speedups on the largest two datasets (UK and PR2).

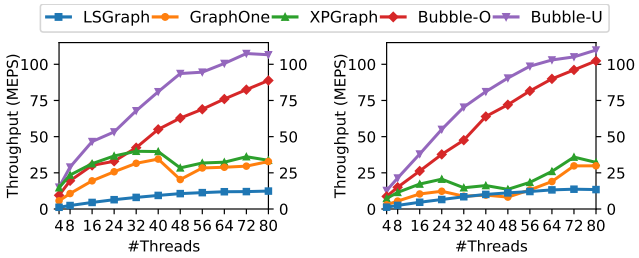
Notably, LSGraph exhibits the slowest loading performance among the competitors because it is the only system among the three competitors that keeps neighbors in sorted order. When inserting an edge, LSGraph must locate the exact position in the neighbor list instead of simply appending it to the neighbor block, as XPGraph and GraphOne do. In contrast, Bubble-O preserves ordered neighbors through sorting, adding only minimal local sorting overhead and thereby providing better insertion performance.

4.3 Graph Analytics Performance

Although Bubble shows excellent ingestion performance, we still need to consider potential trade-offs in its graph analysis performance. To this end, we evaluate the graph analysis performance of Bubble on four common graph algorithms. As described in §4.1, the implementations of these algorithms are consistent across all systems. We test on all datasets to assess the systems' performance on graphs of different sizes and distributions.

First, we evaluate the analytics performance of the systems on BFS, which is a fundamental graph algorithm that many graph analytics workloads rely on [63]. The results are shown in Figure 12a. Bubble-U and Bubble-O still achieve the best performance, reducing the running time by 49.0%, 34.5%, and 30.7% compared to LSGraph, XPGraph, and GraphOne, respectively. The average running time of Bubble-O is 12% lower than that of Bubble-U, mainly due to better locality from sorting neighbors by destination nodes.

Figure 12b shows the results for the PageRank algorithm. The average performance of Bubble is still the best. PageRank traverses all edges of the graph in each iteration. The horizontal scanning optimization employed in Bubble successfully converts the access pattern into sequential reads within sorted segments. For uniform and large datasets (such as PR2), the performance of all systems



(a) Twitter. (b) Friendster.

Figure 13: Scalability of Ingestion with Multi-Threading.

is close, owing to the good locality of the PageRank algorithm. XPGraph achieves the best performance on PR2 (82.9% running time compared to Bubble-O), possibly because its dynamic arrays store the neighbors of each vertex contiguously, thus improving locality in memory access and computation. Note that LSGraph performs poorly on the most skewed graph datasets (LJ, TW) in BFS and PageRank, likely due to the overhead of deeper tree structures for high-degree vertices.

Figure 12c presents the results of the CC algorithm. LSGraph performs well on smaller or skewed datasets (77.2% running time compared to Bubble-O on TW), while Bubble-O is faster on larger or more uniform datasets (66.4% running time compared to LSGraph on PR2). XPGraph and GraphOne perform poorly on all datasets. The main reason for this difference is that the CC algorithm consists of two stages: sampling and finalization. In the sampling step, the algorithm selects the first two neighbors of each vertex and derives an approximate maximum connected component. In finalization, it skips vertices in the approximate maximum connected component and computes the connected components of the remaining smaller vertices. LSGraph stores a fixed number of neighbors for each vertex in contiguous arrays, making the sampling phase faster. However, on larger or more uniform graphs, the finalization phase takes a higher proportion of the time, and Bubble achieves higher efficiency.

We also compare the performance of Bubble and LSGraph on the Triangle Counting (TC) algorithm to assess Bubble-O on workloads that require ordered neighbor interfaces. Figure 12d shows that Bubble-O beat LSGraph on small or more uniform datasets (LJ, PR1, PR2), while LSGraph performs better on skewed datasets (TW, FR, UK). In further analysis, we found that most of the performance gap in skewed datasets is caused by the different parallel libraries. LSGraph employs OpenCilk [64], which provides a work-stealing runtime scheduler that is more effective under imbalanced workloads. Unfortunately, OpenCilk is bound to a custom clang compiler, so it is hard to migrate Bubble to it for further testing.

4.4 Scalability

To validate that the design of Bubble improves the scalability of graph ingestion, we evaluate the update throughput of competitor systems using different thread counts. As shown in Figure 13, we ingest the Twitter and Friendster datasets with 4 to 80 threads. On both datasets, Bubble shows the best scalability, and Bubble-O achieves nearly linear speedups with the number of threads. Bubble-U also exhibits good scalability but encounters a memory bandwidth bottleneck once the throughput exceeds 100 MEPS. As a result, Bubble-O and Bubble-U show similar performance with 80 threads (see results in Figure11), even though Bubble-O has a heavier sorting workload.

LSGraph has a smooth performance curve but is significantly slower than the other systems and does not exhibit good scalability after exceeding 48 threads. XPGraph and GraphOne show similar trends, with two turning points in their scalability curves (32 and 56 threads on Twitter, 24 and 72 threads on Friendster). According to our analysis, the first scalability drop is caused by their single-threaded dispatcher design (§2.3.3), while the second drop results from LLC contention (§2.3.1) and thread imbalance (§2.3.2). Compared to Friendster, the Twitter dataset is more skewed, which increases the cache hit rate during dispatching but also increases workload imbalance. Therefore, on the Twitter graph, the first turning point occurs later, and the second emerges earlier than on Friendster.

Table 2: Memory Usage in GB.

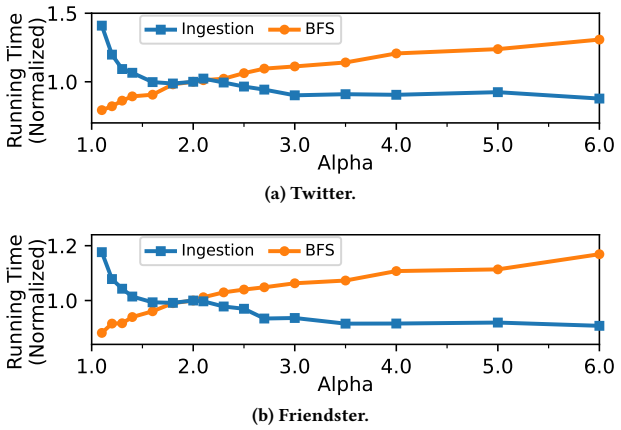
	Bubble-O	Bubble-U	LSGraph	XPGraph	GraphOne
LJ	1.59	**1.58**	2.42	2.45	2.74
PR1	**19.20**	19.21	29.25	22.31	20.04
TW	**26.01**	26.19	44.57	33.29	33.77
FR	**43.46**	43.86	75.11	49.75	57.85
UK	**66.15**	66.28	120.54	73.36	88.98
PR2	70.16	70.18	125.25	**55.54**	74.84

4.5 Memory Footprint

Table 2 reports the memory usage of each system on different datasets. We measure the resident memory after ingesting the graph and running BFS on each dataset. Remember that Bubble stores both source and target vertex IDs in the edge array to sort edges correctly, while other systems only store target vertex IDs, resulting in theoretically twice the memory overhead. However, in our evaluation, Bubble achieves the highest memory efficiency on 5 of the 6 datasets, consuming 54.98%–65.66%, 64.52%–126.38%, and 57.58%–95.82% of the memory used by LSGraph, XPGraph, and GraphOne, respectively. Bubble-O and Bubble-U use nearly the same amount of memory, since the only difference between them is the comparison function used in sorting.

The low memory usage of Bubble is primarily due to its compact data structure, wherein all edges in MuSEA are stored contiguously in a single array with no reserved space for future insertions. All three compared systems reserve space for each vertex to accelerate ingestion. For instance, LSGraph and XPGraph allocate small arrays for each vertex to store the first few neighbors, and their PMAs maintain gaps between neighbors. GraphOne and XPGraph allocate memory in fixed-size blocks, resulting in internal fragmentation.

In dense graphs such as PR2, Bubble consumes more memory due to the overhead of segment adaptive indexes. Bubble allocates index space for each segment using a fixed proportion of the space occupied by the segment (1/8 in our experiments). For denser graphs, this allocation can be larger than the per-vertex index used by other systems. Users can manually adjust the proportion of index space to reduce the memory overhead.



(a) Twitter.

(b) Friendster.

Figure 14: Ingestion and BFS Performance with Different Merging Efficiency Threshold α.

4.6 Sensitivity of Merging Efficiency Threshold

The merging efficiency threshold, α, is a key parameter in Bubble. As analyzed in §3.3.3, adjusting α changes the merging frequency, creating a trade-off between analytics and ingestion performance. We evaluated Bubble's performance on the Twitter and Friendster datasets with different α values. The results in Figure 14 confirm our theoretical analysis: a lower α leads to more frequent merges, improving analytics performance at the cost of ingestion throughput, while a higher α has the opposite effect.

Moreover, the experiments showed that the performance of Bubble was stable around the default configuration ($\alpha = 2.0$), but the observed curve is smoother than theory suggests. For example, in the Friendster dataset, raising α from 2.0 to 4.0 improves update bandwidth by only 11.7% while increasing BFS running time by only 13.4%. This discrepancy occurs because the theoretical analysis limits the worst-case bounds, whereas in practice, both the segment counts and write amplification are far below these bounds.

5 Related Works

In-memory evolving graph processing. Our work focuses on in-memory evolving graph processing. Numerous approaches have been proposed in this field, achieving remarkable progress [5, 23–25, 27, 29–31, 35, 35, 47, 52, 58, 72, 75]. Similar to our target, some studies also emphasize the batch-ingestion efficiency of dynamic graphs [25, 39, 40, 47, 58, 61]. GraphOne [39] and GastCoCo [47] batch edges updates and reserve space in adjacency lists, while Terrace [58] and LSGraph [61] organize data in tree-based structures, mixing formats like edge arrays and PMA to accelerate ingestion and analytics. While these techniques have improved ingestion performance, but in such high-speed scenarios, cache contention and load imbalance become more significant, and limit the scalability of graph systems. We address scalability issues by introducing a low-contention graph format, MuSEA, and a new parallel ingestion strategy, thereby delivering better multi-core performance.

Some other in-memory dynamic graph systems focus on incremental computation [25, 29, 35, 52, 70]. While has a different target, they use similar data structures as other evolving systems, such as adjacency lists [29, 52] or tree-based designs [25, 35], which face the same scalability issues, and our solutions has potential to be applied to them as well.

Out-of-core graph processing. Out-of-core graph systems [4, 28, 36, 38, 43, 49, 63, 68, 74, 79, 80, 82] store graph data in disks, and can handle graphs that exceed main memory. Some out-of-core systems support dynamic graph processing [21, 32, 37, 51, 65, 83, 84]. These systems focusing on I/O efficiency rather than parallel performance. However, with faster storage devices such as persistent memory [2] and NVMe SSDs, more cores are required to fully leverage device throughput [26, 33], so parallel optimizations in Bubble could also be beneficial for out-of-core systems as well.

Edge-centric systems are another out-of-core approach [62, 77, 85], which store the graph as raw edge lists and transform analytics into log-file traversals. Although they achieve high I/O throughput, these designs often suffer from poorer analytics performance and limited programmability [77]. Bubble also appends the edges first, but by adopting a sorting-based strategy, it supports the vertex-centric query, which is more efficient and easier to use. LSMGraph [81] and MyRocks [54] use LSM-trees [57] to convert small I/O operations into bulk writes by building sequential tables. Bubble also exploits the ordered structure in MuSEA, but we further explore and address parallelism challenges in in-memory systems.

Distributed graph processing. There are also many graph systems designed for distributed environments [12, 17–19, 46]. These systems mainly focus on optimizing partitioning and communication to manage large-scale graphs across multiple machines.

In distributed settings, cross-node communication latency tends to be the primary bottleneck rather than inter-core contention. Therefore, the solution proposed in Bubble may not be directly applicable. A more suitable approach might be to use Bubble as a local graph storage engine and build a distributed layer on top of it. Nonetheless, we expect Bubble to perform well in disaggregated memory architectures, such as those based on NUMA, RDMA, or CXL, because these architectures offer high throughput with higher latency in accessing a large remote memory pool, similar to local memory access in Bubble.

6 Conclusion

In this paper, we develop Bubble, a novel evolving graph system that supports highly scalable graph ingestion and efficient analytics. By introducing the MuSEA format, Bubble updates numerous partitions in parallel and minimizes mutual interference, which takes full advantage of modern multi-core processors. We also design an efficient adaptive index structure and several query optimizations for the noncontiguous edge storage in MuSEA to minimize query overhead. Our experiments show that Bubble significantly outperforms cutting-edge systems in graph ingestion while maintaining comparable or higher graph analytics performance.

Acknowledgments

This work was supported in part by NSFC (62472392, 62172382) and the Youth Innovation Promotion Association CAS. AI tools was used to check spelling and grammar errors and polish the text, but no paragraph of this paper was generated by AI alone.

References

[1] 2025. AMD EPYC™ 9965. https://www.amd.com/en/products/processors/server/epyc/9005-series/amd-epyc-9965.html

[2] 2025. Intel® Optane™ Persistent Memory (PMem). https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html

[3] 2025. Intel® Xeon® 6944P Processor - Product Specifications. https://www.intel.com/content/www/us/en/products/sku/242856/intel-xeon-6944p-processor-432m-cache-1-80-ghz/specifications.html

[4] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2019. Clip: A Disk I/O Focused Parallel Out-of-Core Graph Processing System. *IEEE Transactions on Parallel and Distributed Systems* 30, 1 (Jan. 2019), 45–62. doi:10.1109/TPDS.2018.2858250 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[5] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. 2022. VCSR: Mutable CSR Graph Format Using Vertex-Centric Packed Memory Array. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid '22)*. 71–80. doi:10.1109/CCGrid54584.2022.00016

[6] Khaled Ammar. 2016. Techniques and Systems for Large Dynamic Graphs. In *Proceedings of the 2016 on SIGMOD'16 PhD Symposium (SIGMOD'16 PhD)*. Association for Computing Machinery, New York, NY, USA, 7–11. doi:10.1145/2926693.2929897

[7] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic Acids Research* 46, 6 (Jan. 2018), e33–e33. doi:10.1093/nar/gkx1313

[8] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '06)*. Association for Computing Machinery, New York, NY, USA, 44–54. doi:10.1145/1150402.1150412

[9] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. doi:10.48550/arXiv.1508.03619 arXiv:1508.03619.

[10] Michael A. Bender and Haodong Hu. 2007. An adaptive packed-memory array. *ACM Trans. Database Syst.* 32, 4 (Nov. 2007), 26–es. doi:10.1145/1292609.1292616

[11] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2021. Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems. *IEEE Transactions on Parallel and Distributed Systems* (2021), 1–1. doi:10.1109/TPDS.2021.3131677 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[12] Maciej Besta, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Nils Blach, Berke Egeli, Georgy Mitenkov, Wojciech Chlapek, Marek Michalewicz, Hubert Niewiadomski, Juergen Mueller, and Torsten Hoefler. 2023. The Graph Database Interface: Scaling Online Transactional and Analytical Graph Workloads to Hundreds of Thousands of Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Association for Computing Machinery, New York, NY, USA, 1–18. doi:10.1145/3581784.3607068

[13] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. 2021. Tesseract: distributed, general graph pattern mining on evolving graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 458–473. doi:10.1145/3447786.3456253

[14] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A large time-aware web graph. *SIGIR Forum* 42, 2 (Nov. 2008), 33–38. doi:10.1145/1480506.1480511

[15] P. Boldi and S. Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web (WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602. doi:10.1145/988672.988752

[16] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC '18)*. 1–7. doi:10.1109/HPEC.2018.8547541 ISSN: 2377-6943.

[17] Andrew Carter, Andrew Rodriguez, Yiming Yang, and Scott Meyer. 2019. Nanosecond Indexing of Graph Data With Hash Maps and VLists. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 623–635. doi:10.1145/3299869.3314044

[18] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. 2022. G-tran: a high performance distributed graph database with a decentralized architecture. *Proc. VLDB Endow.* 15, 11 (July 2022), 2545–2558. doi:10.14778/3551793.3551813

[19] Hongzhi Chen, Bowen Wu, Shiyuan Deng, Chenghuan Huang, Changji Li, Yichao Li, and James Cheng. 2020. High Performance Distributed OLAP on Property Graphs with Grasper. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2705–2708. doi:10.1145/3318464.3384685

[20] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 85–98. doi:10.1145/2168836.2168846

[21] Yongli Cheng, Yan Ma, Hong Jiang, Lingfang Zeng, Fang Wang, Xianghao Xu, and Yuhang Wu. 2024. TgStore: An Efficient Storage System for Large Time-Evolving Graphs. *IEEE Transactions on Big Data* 10, 02 (April 2024), 158–173. doi:10.1109/TBDATA.2024.3366087 Publisher: IEEE Computer Society.

[22] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: graph processing at Facebook-scale. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1804–1815. doi:10.14778/2824032.2824077

[23] Dean De Leo and Peter Boncz. 2021. Teseo and the analysis of structural dynamic graphs. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1053–1066. Publisher: VLDB Endowment.

[24] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-trees: supporting parallel and compressed purely-functional collections. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 108–121. doi:10.1145/3519939.3523733

[25] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation (PLDI '19)*. 918–934.

[26] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR '22)*. Association for Computing Machinery, New York, NY, USA, 120–127. doi:10.1145/3534056.3534945

[27] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing (HPEC '12)*. 1–5.

[28] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. 2019. Large-Scale Graph Processing on Emerging Storage Devices. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*. 309–316. https://www.usenix.org/conference/fast19/presentation/elyasi

[29] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. 513–527.

[30] Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Dalila Chiadmi, Sungpack Hong, and Hassan Chafi. 2020. CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure. *24th International Conference on Principles of Distributed Systems (OPODIS'20)* (2020). doi:10.4230/LIPIcs.OPODIS.2020.17

[31] Per Fuchs, Domagoj Margan, and Jana Giceva. 2022. Sortledton: a universal, transactional graph data structure. *Proceedings of the VLDB Endowment* 15, 6 (Feb. 2022), 1173–1186. doi:10.14778/3514061.3514065

[32] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar storage and list-based processing for graph database management systems. *Proceedings of the VLDB Endowment* 14, 11 (July 2021), 2491–2504. doi:10.14778/3476249.3476297

[33] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. *Proceedings of the VLDB Endowment* 16, 9 (May 2023), 2090–2102. doi:10.14778/3598581.3598584

[34] Abdullah Al Raqibul Islam and Dong Dai. 2023. DGAP: Efficient Dynamic Graph Analysis on Persistent Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3581784.3607106

[35] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. 2021. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*. 337–355. https://www.usenix.org/conference/nsdi21/presentation/iyer

[36] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*. 411–424. doi:10.1109/ISCA.2018.00042 ISSN: 2575-713X.

[37] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. 2017. ZipG: A Memory-efficient Graph Store for Interactive Queries. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1149–1164. doi:10.1145/3035918.3064012

[38] Juno Kim and Steven Swanson. 2022. Blaze: Fast Graph Processing on Fast SSDs. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '22)*. 1–15. doi:10.1109/SC41404.2022.00049 ISSN: 2167-4337.

[39] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*. 249–263.

[40] Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)* 15, 4 (2020), 1–40. Publisher: ACM New York, NY, USA.

[41] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web (WWW '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1343–1350. doi:10.1145/2487788.2488173

[42] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web (WWW '10)*. Association for Computing Machinery, New York, NY, USA, 591–600. doi:10.1145/1772690.1772751

[43] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. 31–46.

[44] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE '13)*. 38–49. doi:10.1109/ICDE.2013.6544812 ISSN: 1063-6382.

[45] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data

[46] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, Xudong Wang, Huiming Zhu, Xuwei Fu, Tingwei Wu, Hongfei Tan, Hengtian Ding, Mengjin Liu, Kangcheng Wang, Ting Ye, Lei Li, Xin Li, Yu Wang, Chenguang Zheng, Hao Yang, and James Cheng. 2022. ByteGraph: a high-performance distributed graph database in ByteDance. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3306–3318. doi:10.14778/3554821.3554824

[47] Hongfu Li, Qian Tao, Song Yu, Shufeng Gong, Yanfeng Zhang, Feng Yao, Wenyuan Yu, Ge Yu, and Jingren Zhou. 2025. GastCoCo: Graph Storage and Coroutine-Based Prefetch Co-Design for Dynamic Graph Processing. *Proc. VLDB Endow.* 17, 13 (Feb. 2025), 4827–4839. doi:10.14778/3704965.3704986

[48] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, Weimin Zheng, and Jingfang Xu. 2018. ShenTu: Processing Multi-Trillion Edge Graphs on Millions of Cores in Seconds. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '18)*. 706–716. doi:10.1109/SC.2018.00059

[49] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-grained IO management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST '17)*. 285–300.

[50] Yao Ma, Ziyi Guo, Zhaocun Ren, Jiliang Tang, and Dawei Yin. 2020. Streaming Graph Neural Networks. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '20)*. Association for Computing Machinery, New York, NY, USA, 719–728. doi:10.1145/3397271.3401092

[51] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering (ICDE '15)*. 363–374.

[52] Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: sparsity-aware incremental processing of streaming graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 83–98. doi:10.1145/3447786.3456230

[53] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 116–128. doi:10.1145/3307650.3322275

[54] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-tree database storage engine serving Facebook's social graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230. Publisher: VLDB Endowment.

[55] Seth A. Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. 2014. Information network or social network? the structure of the twitter follow graph. In *Proceedings of the 23rd International Conference on World Wide Web (WWW '14 Companion)*. Association for Computing Machinery, New York, NY, USA, 493–498. doi:10.1145/2567948.2576939

[56] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 456–471. doi:10.1145/2517349.2522739

[57] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[58] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1372–1385. doi:10.1145/3448016.3457313

[59] Orson R. L. Peters. 2021. Pattern-defeating Quicksort. doi:10.48550/arXiv.2106.05123 arXiv:2106.05123 [cs].

[60] Tim Peters. 2002. [Python-Dev] Sorting. https://mail.python.org/pipermail/python-dev/2002-July/026837.html

[61] Hao Qi, Yiyang Wu, Ligang He, Yu Zhang, Kang Luo, Minzhi Cai, Hai Jin, Zhan Zhang, and Jin Zhao. 2024. LSGraph: A Locality-centric High-performance Streaming Graph Engine. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 33–49. doi:10.1145/3627703.3650076

[62] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. 472–488. doi:10.1145/2517349.2522740

[63] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 420–431. doi:10.1145/3186728.3164139

[64] Tao B. Schardl and I-Ting Angelina Lee. 2023. OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 189–203. doi:10.1145/3572848.3577509

[65] Sijie Shen, Zihang Yao, Lin Shi, Lei Wang, Longbin Lai, Qian Tao, Li Su, Rong Chen, Wenyuan Yu, Haibo Chen, Binyu Zang, and Jingren Zhou. 2023. Bridging the Gap between Relational OLTP and Graph-based OLAP. In *2023 USENIX Annual Technical Conference (ATC '23)*. 181–196. https://www.usenix.org/conference/atc23/presentation/shen

[66] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 135–146. doi:10.1145/2442516.2442530

[67] Jaewoong Sim, Jaekyu Lee, Moinuddin K. Qureshi, and Hyesoon Kim. 2012. FLEXclusion: balancing cache capacity and on-chip bandwidth via flexible exclusion. *SIGARCH Comput. Archit. News* 40, 3 (June 2012), 321–332. doi:10.1145/2366231.2337196

[68] Keval Vora. 2019. LUMOS: Dependency-Driven Disk-based Graph Processing. In *2019 USENIX Annual Technical Conference (ATC '19)*. 429–442.

[69] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic Analysis of Evolving Graphs. *ACM Transactions on Architecture and Code Optimization* 13, 4 (Oct. 2016), 32:1–32:27. doi:10.1145/2992784

[70] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems (ASPLOS '17)*. 237–251.

[71] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18)*. Association for Computing Machinery, New York, NY, USA, 839–848. doi:10.1145/3219819.3219869

[72] Rui Wang, Shuibing He, Weixu Zong, Yongkun Li, and Yinlong Xu. 2022. XP-Graph: XPline-Friendly Persistent Memory Graph Stores for Large-Scale Evolving Graphs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*. 1308–1325. doi:10.1109/MICRO56248.2022.00091

[73] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. 2020. Graphwalker: An i/o-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *2020 USENIX Annual Technical Conference (ATC '20)*. 559–571.

[74] Rui Wang, Weixu Zong, Shuibing He, Xinyu Chen, Zhenxin Li, and Zheng Dang. 2024. Efficient Large Graph Processing with Chunk-Based Graph Representation Model. In *2024 USENIX annual technical conference (USENIX ATC 24)*. 1239–1255. https://www.usenix.org/conference/atc24/presentation/wang-rui

[75] Brian Wheatman and Helen Xu. 2018. Packed Compressed Sparse Row: A Dynamic Graph Representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC) (HPEC '18)*. 1–7. doi:10.1109/HPEC.2018.8547566 ISSN: 2377-6943.

[76] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. faimGraph: High Performance Management of Fully-Dynamic Graphs Under Tight Memory Constraints on the GPU. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '18)*. 754–766. doi:10.1109/SC.2018.00063

[77] Xianghao Xu, Fang Wang, Hong Jiang, Yongli Cheng, Dan Feng, and Yongxuan Zhang. 2020. A Hybrid Update Strategy for I/O-Efficient Out-of-Core Graph Processing. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (Aug. 2020), 1767–1782. doi:10.1109/TPDS.2020.2973143 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[78] Ke Yang, Xiaosong Ma, Saravanan Thirumuruganathan, Kang Chen, and Yongwei Wu. 2021. Random Walks on Huge Graphs at Cache Efficiency. In *Proceedings*

of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21). Association for Computing Machinery, New York, NY, USA, 311–326. doi:10.1145/3477132.3483575

[79] Tsun-Yu Yang, Yizou Chen, Yuhong Liang, and Ming-Chang Yang. 2024. Seraph: Towards Scalable and Efficient Fully-external Graph Computation via On-demand Processing. In *22nd USENIX Conference on File and Storage Technologies (FAST '24)*. 373–387. https://www.usenix.org/conference/fast24/presentation/yang-tsun-yu

[80] Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. 2022. Practicably Boosting the Processing Performance of BFS-like Algorithms on Semi-External Graph System via I/O-Efficient Graph Ordering. In *20th USENIX Conference on File and Storage Technologies (FAST '22)*. 381–396. https://www.usenix.org/conference/fast22/presentation/yang

[81] Song Yu, Shufeng Gong, Qian Tao, Sijie Shen, Yanfeng Zhang, Wenyuan Yu, Pengxi Liu, Zhixin Zhang, Hongfu Li, Xiaojian Luo, Ge Yu, and Jingren Zhou. 2024. LSMGraph: A High-Performance Dynamic Graph Storage System with Multi-Level CSR. *Proc. ACM Manag. Data* 2, 6 (Dec. 2024), 243:1–243:28. doi:10.1145/3698818

[82] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *13th USENIX conference on file and storage technologies (FAST '15)*. 45–58.

[83] Long Zheng, Xiangyu Ye, Haifeng Liu, Qinggang Wang, Yu Huang, Chuangyi Gui, Pengcheng Yao, Xiaofei Liao, Hai Jin, and Jingling Xue. 2023. AFaVS: Accurate Yet Fast Version Switching for Graph Processing Systems. In *2023 IEEE 39th International Conference on Data Engineering (ICDE '23)*. 53–66. doi:10.1109/ICDE55515.2023.00012 ISSN: 2375-026X.

[84] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proceedings of the VLDB Endowment* 13, 7 (March 2020).

[85] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (ATC '15)*. 375–386. https://www.usenix.org/conference/atc15/technical-session/presentation/zhu

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

This paper presents Bubble, a scalable evolving graph system for modern multi-core processors. Our key contributions can be summarized as follows:

- **C_1** The MuSEA, a graph format allowing a large number of subgraphs to be updated in parallel and minimizing cache contention among threads.
- **C_2** The Bubble parallel ingestion policy, featuring a lock-free, zero-copy dispatcher and a work-stealing-based scheduler.
- **C_3** The query optimizer in Bubble, which allows accessing edges in MuSEA with customizable patterns to adapt to the requirements of various graph algorithms.
- **C_4** Comparison of Bubble against state-of-the-art systems, including GraphOne, XPGraph, and LSGraph, on a set of real-world graph datasets and common graph algorithms.

A.2 Computational Artifacts

Two computational artifacts are archived under a single DOI, but we recommend using the two GitHub repositories for better organization and ease of use.

Repo1: https://github.com/ldeng-ustc/bubble
Repo2: https://github.com/ldeng-ustc/bubble_rel
DOI: 10.5281/zenodo.15781200

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1,C_2,C_3,C_4	Figures 11-13 Table 2
A_2	C_4	Figures 11-13 Table 2

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

This artifact contains the main implementation of Bubble, which combines three key designs: the MuSEA (C_1), the parallel ingestion policy (C_2), and the query optimizer (C_3). The artifact is provided as a C++ library, and Bubble in the comparison against other systems (C_4) is built on top of this library. We provide an example code to verify the library.

Expected Results

The example program should compile and run successfully.

Expected Reproduction Time (in Minutes)

The expected building time for this artifact is 10 min. The expected execution time of the example program is less than 1 min.

Artifact Setup (incl. Inputs)

Hardware. This artifact has no specific hardware requirements, but to reproduce the results, a dual-socket server with at least 80 logical cores, 128 GB of RAM and at least 300 GB of storage is recommended. A network connection is required for downloading the dependencies and datasets.

Software. The artifact depends on several libraries; these libraries will be automatically downloaded during compilation, and **there is no need to install them manually**. These libraries include: numactl (v2.0.18), cxxopts (v3.2.0), indicators (v2.3), hwloc (v2.11.2), fmtlib (v10.1.1), parlay (v2.3.1), concurrentqueue (v1.0.4), and readerwriterqueue (v1.0.6).

Datasets / Inputs. This artifact requires no additional inputs.

Installation and Deployment. The artifact should be compiled and run on a Linux-based system. The requirements for the system and compilers are listed below.

- Kernel: >= 5.15.0
- C++ compiler: g++ 11.4.0
- CMake: >= 3.20.0
- Boost: >= 1.83.0

We recommend reproducing our work on Ubuntu 24.04 because the current default versions of the software in its package manager meet all requirements, avoiding complex manual installations.

Artifact Execution

The workflow consists of two tasks: *compiling* and *running*. In *compiling*, the library and the example code are built using CMake, and the example program is generated. Then, in *running*, the example program is executed, and the output is generated.

Artifact Analysis (incl. Outputs)

No extra processing is required for this artifact and its output. To verify the artifact, check whether the example program finishes running successfully (with return code 0) and that there are no error messages in its output.

B.2 Computational Artifact A_2

Relation To Contributions

This artifact provides implementations for common graph algorithms using Bubble and three competitive systems (GraphOne, XPGraph, and LSGraph). The graph algorithms include BFS, PageRank, CC, and TC.

For comparison (C_4), the artifact also includes corresponding testing programs and running scripts for all four systems. Forked versions of the three competitors are provided in the artifact, based on the main branch of their codebases. The testing programs load the graph data, run the algorithms, and collect ingestion and analysis time, memory usage, and other runtime metrics for each system.

The artifact also provides plotting scripts to process the output of the testing programs and generate Figures 11-13 in the paper.

Expected Results

The testing programs and plotting scripts should run successfully, and the output should be similar to the figures in the paper, indicating the following conclusions:

- The ingestion throughput of Bubble is significantly higher than the other systems (> 2.5×).
- The running time of Bubble for BFS, PageRank, and CC is shorter than or similar to the other systems (< 150%).
- In terms of TC running time, Bubble and LSGraph outperform each other on different datasets.
- Bubble shows better scalability than the other systems in graph ingestion.
- The memory usage of Bubble is lower than the other systems in most cases.

Expected Reproduction Time (in Minutes)

The expected building time for this artifact is 10 min. The expected dataset preparation time is 1 hour (including downloading and preprocessing). The expected execution time of the testing programs is about 6 hours (including running all algorithms on all datasets, with each test run only once). The expected plotting time is 5 min.

Artifact Setup (incl. Inputs)

Hardware. Any server with at least 80 logical cores, 128 GB of RAM, and at least 300 GB of storage is sufficient to reproduce the results. Our experiments were conducted on a dual-socket Dell R740 server with two Intel Xeon Gold 5218R CPUs at 2.10 GHz. Each processor has 20 physical cores and 40 logical cores with hyper-threading enabled. Each CPU has 28 MB of L3 cache. The L2 and L1 caches are 1 MB and 64 KB per core, respectively. The server is equipped with 220 GB of DRAM and 1.5 TB of NVMe SSD storage.

Software. The software requirements for Bubble are the same as for A_1. However, the competing systems have the following additional requirements:

- OpenCilk (v1.0)
  https://github.com/OpenCilk/opencilk-project/releases
- parlaylib (v2.3.1, must be installed manually)
  https://github.com/cmuparlay/parlaylib

Additionally, running the testing and plotting scripts requires Python 3.10 or higher. The plotting scripts also depend on the following Python packages:

- matplotlib (v3.8.2)
- numpy (v1.26.2)
- pandas (v2.2.2)
- seaborn (v0.13.2)

Datasets / Inputs. The experiments use six real-world graph datasets: LiveJournal, Twitter, Protein2, Protein3, UK2007, and Friendster. Scripts are provided to download and preprocess these datasets.

Installation and Deployment. The requirements for installation and deployment are the same as for A_1.

Artifact Execution

The workflow consists of four tasks: *compiling*, *data preparation*, *running*, and *plotting*. The workflow dependency is: *compiling* → *data preparation* → *running* → *plotting*.

- The *compiling* task builds the four systems (Bubble and the three competitors) using CMake and Make, generating the testing programs.
- The *data preparation* task downloads the datasets, preprocesses them, converts them into binary edge lists, and shuffles them.
- The *running* task executes the testing programs, running all algorithms on all datasets for all systems and writing the output results to files.
- Finally, the *plotting* task processes the output files and generates the figures.

Artifact Analysis (incl. Outputs)

The plotting scripts process the output from the *running* task to generate Figures 11-13 in the paper. To generate Table 2, run the provided memory usage script and check its standard output.

Artifact Evaluation (AE)

C.1 Computational Artifact A_1

Artifact Setup (incl. Inputs)

Compiling and installing Bubble is straightforward. All dependencies, except for Boost, are automatically downloaded and built. Before starting, ensure you have a complete Boost installation and the required compiler toolchain (see AD for versions).

We recommend using Ubuntu 24.04 for easy access to the necessary tools. The following steps use this system as an example:

(1) Install the build toolchain and Boost. On Ubuntu 24.04, the default versions meet the requirements. On older systems, you may need to install the correct versions manually.

```
sudo apt update
sudo apt install build-essential cmake libboost-all-dev
```

(2) Download the Bubble source code into your workspace directory. You can use Git or download it from the Zenodo archive.

```
git clone https://github.com/ldeng-ustc/bubble.git
```

(3) Enter the source directory and compile using CMake.

```
cd bubble
cmake -S all -B build
cmake --build build -j
```

This will compile Bubble and its test programs into the `build` directory. To run other algorithms with Bubble, you can add your code to the `app` directory (or a subdirectory) and re-run the compilation commands. If you installed Boost manually, specify its path by adding `-DBoost_ROOT=/path/to/your/boost` to the `cmake` command.

Artifact Execution

The complete experimental workflow, which includes competing systems, is described under Artifact A_2. This section only covers how to run the standalone Bubble benchmark. **Note: You do not need to perform these steps if your goal is to reproduce the full results from the paper.**

The basic test program, `bubble/build/app/benchmarks`, is generated during the setup process. To run it, you must specify an input dataset with the `-f` flag and the number of threads with the `-t` flag. The program will load the dataset, run the BFS, PageRank, and CC algorithms, and output their execution times and other metrics.

An ordered version of the benchmark, `benchmarks_ordered`, is also provided and uses the same arguments.

We provide a script to simplify execution:

```
./script/bubble/basic_benchmarks.sh <dataset> \
    <vertex_count> <stdout_file> <ingest_thread>
```

The output will be redirected to `<stdout_file>`.

For example, to run the benchmark on the example K20 graph and save the output to `output.txt`, use:

```
./script/bubble/basic_benchmarks.sh \
    ./example_data/K20/edgelist.bin 20 output.txt 40
```

Artifact Analysis (incl. Outputs)

Artifact A_1 does not include a separate analysis step. All procedures for analyzing results and generating figures are described in Artifact A_2.

C.2 Computational Artifact A_2

Artifact Setup (incl. Inputs)

We provide a repository containing the source code and test programs for all systems evaluated in the paper (Bubble and its competitors). The setup involves installing dependencies, downloading the code, and running a build script.

(1) **Install Dependencies.** On Ubuntu 24.04, you can run:

```
sudo apt update
sudo apt install libz3-dev libpmemobj-dev libtbb-dev \
    numactl libnuma-dev libstdc++-14-dev
```

(2) **Build Competing Systems.** We have created a script to build all three competing systems. Download the repository to the same directory as Bubble (i.e., your workspace, the parent directory of `bubble`).

```
git clone https://github.com/ldeng-ustc/bubble_rel.git
mv bubble_rel graph_works
cd graph_works
./build_all.sh
```

The `build_all.sh` script navigates to the `GraphOne`, `XPGraph`, and `LSGraph` source directories and compiles them to generate executable benchmark programs. It also automatically downloads the required version of OpenCilk and sets it up to compile LSGraph.

(3) **Link Repositories.** For the experiment scripts to function correctly, you must create a symbolic link from the `graph_works` directory to the Bubble source directory.

```
# Both bubble and graph_works should be in the same
# directory (i.e., your workspace)
cd ../bubble # Go to bubble source directory
ln -s ../graph_works ./related_works
```

Note: When compiling LSGraph, you might encounter a missing `iostream` error. This typically requires installing the `libstdc++` development package corresponding to the g++ version used by clang. You can find the g++ version after step 2 by running command below in the `LSGraph` subdirectory:

```
cd graph_works/LSGraph
./opencilk/OpenCilk-10.0.1-Linux/bin/clang++ -v -E
```

And check the output about "Selected GCC installation". For example, if the output shows clang selected g++ 11, you need to install the `libstdc++-11-dev` package, by:

```
sudo apt install libstdc++-11-dev
```

And then re-run the step 2 and step 3 above to rebuild LSGraph and create the symbolic link.

Artifact Execution

To reproduce the paper's results, you must first prepare the datasets and then run the experiment scripts. **The uncompressed and shuffled datasets are quite large (over 300 GB), so ensure you have sufficient disk space in your workspace.**

(1) **Download Datasets.** Download `bubble-datasets.tar.gz` from the provided Zenodo repository. The datasets are compressed in the WebGraph format and need to be decompressed and shuffled.

To download the datasets, use the following command (note that this is a single-line command, the long URL is break for better readability):

```
# In your workspace directory
# About 13.6 GB
wget -O bubble-datasets.tar.gz "https://zenodo.org/records
    /15781200/files/bubble-datasets.tar.gz?download=1"
```

(2) **Prepare Datasets.** First, install the necessary tools for processing.

```
sudo apt update
sudo apt install rustup libboost-all-dev
rustup default stable
```

Next, download and build our modified version of `webgraph-rs` into your workspace for converting the datasets into the required format.

```
git clone https://github.com/ldeng-ustc/webgraph-rs.git
cd webgraph-rs
cargo build --release
```

Finally, use the provided script in the Bubble repository to automate the entire data preparation process. It unpacks the `bubble-datasets.tar.gz` file, converts the datasets to binary edge lists (both directed and undirected versions), shuffles them, and creates symbolic links in the correct locations. **This step will take about 1-2 hours, and the final datasets will occupy about 300 GB of disk space.**

If you have already run the basic benchmark in execution step (section C.1) of A_1, you must first delete the existing `data/` directory in the Bubble source directory to avoid conflicts.

```
cd .. # Go back to your workspace
cd bubble
# Make sure there is no existing data/ directory
# You can also delete it if you don't need it
mv data/ data_backup/
# Make sure webgraph-rs/target/release is in your
# workspace or specify the path in the script
python3 ./script/generate_datasets_from_webgraph.py
```

If your downloaded file is not in your workspace (the parent directory of `bubble`), you can specify the path by adding the argument `-i /path/to/bubble-datasets.tar.gz` to the script command above.

(3) **Run Experiments.** The `script/` directory contains Python scripts to orchestrate the experiments. First, install the required Python libraries.

```
sudo apt update
sudo apt install python3-numpy python3-pandas \
    python3-matplotlib
```

Then, run the scripts to execute the experiments. (All scripts should be run from the `bubble/` directory.)

- **Basic Performance (Figures 11-12a,b,c, Table 2):** Tests ingestion and algorithm performance using all cores. (Takes about 1-2 hours depending on hardware.)

```
python3 ./script/basic_benchmarks.py
```

- **Scalability (Figure 13):** Tests performance with varying thread counts. (**Takes about 6-10 hours.** To reduce the experimental time, you can use `scalability_fast.py`, which contains only half the data points of the version in the paper.)

```
python3 ./script/scalability.py
# Or a faster version
# python3 ./script/scalability_fast.py
```

- **Triangle Counting (Figure 12d):** Runs TC on Bubble and LSGraph. (Takes about 1-2 hours.)

```
python3 ./script/run_tc.py
```

All results will be saved in a new subdirectory in bubble source, inside `data/experiments/`, named with the execution timestamp.

Warning: The full suite of experiments takes approximately 4-8 hours. We strongly recommend using a terminal multiplexer like `screen` or `tmux` to prevent interruptions. Some related works can be unstable during the analysis phase and may occasionally fail, resulting in missing data points.

Artifact Analysis (incl. Outputs)

The `script/` directory contains plotting scripts that process the experimental data from `data/experiments/` to generate the figures from the paper.

First, create a directory to store the figures (as before, run following commands from the `bubble/` directory):

```
mkdir figs
```

Run the following scripts to generate the corresponding figures:

- **Figures 11, 12(a-c):** Ingestion throughput, algorithm performance.

```
python3 script/2-exp-1-basic_benchmarks-plot.py
```

- **Figure 12(d):** TC algorithm performance, only for Bubble and LSGraph.

```
python3 script/2-exp-2-tc-plot.py
```

- **Figure 13:** Scalability of ingestion.

```
python3 script/2-exp-3-scalability-plot.py
```

- **Table 2:** Memory usage. (Note that the results will print to standard output, as a pandas data frame and a LaTeX table.)

```
python3 script/2-exp-4-memory-print.py
```

After running the scripts, the `figs/` directory will contain plots corresponding to the figures in the paper.

The reproduced figures may not be identical to the paper's figures due to:

(1) **Hardware Differences:** Different machine configurations will lead to different absolute performance values, which may affect plot scaling. As a result, lines or bars may break the boundaries of the plots.

(2) **Instability of Competing Systems:** Data points for the competing systems in Figure 12 may be missing or appear unusually low due to occasional runtime errors. Check the output files in `data/experiments/` for the corresponding error messages.

(3) **Single Run Variance:** To reduce reproduction time, each experiment is run only once. The paper's results are averaged over 5-10 runs. This may lead to higher variance in the reproduced plots, especially for metrics with short execution times.

Despite these variations, the reproduced results should still clearly demonstrate the paper's main conclusions:

- Bubble achieves at least 2× (and on average over 3×) the ingestion throughput of competing systems.
- Bubble's analytics performance is better than or comparable (greater than 0.5×, on average over 1.0×) to that of competitors across most datasets and algorithms.
- Bubble demonstrates near-linear scalability up to 80 cores for graph ingestion.

Reproducibility Report

D Overview of Reproduction of Artifacts

The following table provides an overview of each computational artifact's reproducibility status. Artifact IDs correspond to those in the AD/AE Appendices.

Artifact ID	Available	Functional	Reproduced
A_1	•	•	•
A_2	•	•	•
Badge awarded	yes	yes	yes

E Reproduction of Computational Artifacts

E.1 Timeline

Artifact evaluation was carried out on August 24-25, 2025.

E.2 Computational Environment and Resources

The experiments conducted for artifact evaluation were performed on Chameleon Cloud CHI@UC on a Intel Xeon Cascadelake node. The node OS was Ubuntu 20.04 which is older than that recommended by the paper authors. This led to several changes being needed on the node, eg installing newer compiler and python versions than those available.

E.3 Details on Artifact Reproduction

The appendix included all the necessry steps in enough detail needed to reproduce the results. There were some minor missing details/errors which the paper's authors addressed promptly. These edits are included in their final version of the reproducabilityAD/AE appendix. The evaluation was done based on the earlier version of the appendix. The runtime for the experiments was 10+ hours, longer than expected. The artifacts, data and scripts needed to reproduce the results take a significant amount of disk space, hundreds of GBs. This made it impossible to create a 'cc-snapshot' and easily restart the evaluation instead of recreating everything from scratch.

The data and figures created as part of this exercise reproduce the main findings of the paper, not the exact numerical results which is to be expected. Figure 12c was not reproduced due to an issue in the plotting script the authors have fixed. The data to create the plot was reproduced and inspection of the updated plotting code indicates the issue is resolved.

Overall the appendix was well organized, clear, and correct making the reproduction process straightforward.

Disclaimer: This Reproducibility Report was crafted by volunteers with the goal of enhancing reproducibility in our research domain. The time period allocated for the reproducibility analysis was constrained by paper notification deadlines and camera-ready submission dates. Furthermore, the compute hours in the shared infrastructure (e.g., Chameleon Cloud) available to the authors of this report were limited and restricted the scope and quantity of experiments in the review phase. Consequently, the inability to reproduce certain artifacts within this evaluation should not be interpreted as definitive evidence of their irreproducibility. Limitations in the time allocated to this review and the compute resources available to the reviewers may have prevented a positive outcome. Furthermore, reviewers assess the reproducibility of the artifacts provided by the authors; however, they are not accountable for verifying that the artifacts support the main claims of the paper.